C/C++ Contributing Editors

The New C: Introducing C99

Randy Meyers

C99 is much like C89, but with more of the same — lots more of the same.

Introduction

You might not have noticed, but a major revision [1] to the ANSI/ISO C Standard, called C99, was approved last December. Also, you might not have noticed, but you might already be using the new C language, or at least parts of it.

The reason for this is that the committee took a pretty conservative approach in adding features to C. Almost all of the new features have been implemented and have proved their worth in existing C implementations. Although no implementation yet supports all of C99, many implementations have supported different parts of C99 for years.

This is good news for C programmers. Perhaps you have been avoiding an extension in your favorite compiler because it was not portable. If that extension is a feature of the new C Standard, you can start using the feature knowing it will spread to other compilers as the industry rolls out C99-compliant compilers.

It almost goes without saying that the new Standard is upwardly compatible with the old. There are a few incompatibilities, but they are very minor, and the committee worked very hard to minimize problems. For example, see the discussion of new keywords below.

Names and History

Programming languages evolve over time, and the usual practice is to refer to a language not only by its name, but also the year in which it was defined. (In lectures five years ago, I could get a laugh by giving some examples of this rule: ALGOL 68, C89, Fortran 77, and Fortran 4. Alas, this geeky humor has a Y2K bug: these days 2004 springs to mind before 1904.) Thus, the new language and the Standard that defines it are called C99. The original C Standard [2] is called either C89 or C90. (ANSI published the document in 1989, but ISO renumbered the sections and published the document in 1990.) There was a minor update [3] to C89 called C95 that you probably did not notice unless you process Japanese, Korean, or Chinese text, since it mostly added more library functions that process wide and multibyte characters. (Java proponents sometimes erroneously claim that Java was the first language to support large character sets. Such support was in Standard C in 1989.)

Perhaps the greatest influence on C99 was the Numerical C Extensions Group, or NCEG. The NCEG was a subcommittee of J11, the ANSI C committee, that started working on a technical report [4] after C89 was finalized. The NCEG Tech Report was not a standard, it was a call for implementations to experiment and gain

experience with a set of well described extensions. The majority of these extensions dealt with numerical programming in C (IEEE arithmetic, complex numbers), but some had more general purpose or promoted optimization (variable length arrays, parallel processing, the **restrict** keyword).

In some cases, NCEG extensions were invented by the subcommittee. In others, vendors brought extensions already implemented in their compilers to the committee for review and feedback. Since the tech report was not a standard, vendors were free to pick and choose which extensions to implement, and to modify the extensions based on customer experience.

This real world experience is very valuable. Language features can interact in surprising ways, and sometimes a language feature will cause a run-time penalty even if the feature is never used. (For example, on some C++ implementations, the mere existence of multiple inheritance as a feature slows down programs that use only single inheritance.) The experimentation with NCEG extensions not only improved the extensions themselves, but also improved their specification, and gave the C committee confidence that the interactions and costs of the language features were known.

The C99 That Isn't

Not all NCEG extensions were added to C99. Perhaps the biggest example is the NCEG parallel-processing support, which was based on the C* Language (pronounced C-Star) from Thinking Machines. The manufacturers of parallel computers have various idiosyncratic extensions to write explicitly parallel programs, and the NCEG Technical Report did not change this. Since there is still little consensus on the best way to program parallel computers, such a feature is not yet suitable for including in Standard C.

In other cases, NCEG extensions were modified when added to C99. The NCEG support for complex numbers included separate imaginary datatypes, such as **double_ imaginary**. The imaginary data types were made optional in C99.

However, the biggest feature considered for, but not included in, C99 did not come from the NCEG, but from C++. For about a year, the committee worked on a subset of C++ object-oriented features. Included in the subset were single (but not multiple) inheritance, virtual functions, member access control (public, private, protected), constructors, and destructors. This mix of features was similar to C++ in the late 1980s.

This resemblance to early C++ was both a plus and a minus. On the positive side, this set of features was responsible for the initial popularity of C++, and the set of features was known to work well together with well understood costs and interactions. On the negative side was the question, "Isn't the natural evolution of the C++ of the 1980s the C++ of the 1990s? If so, what is the value in C starting down that path since the C++ of the 1990s already exists?" Ultimately, for a variety of reasons, some logistical, the committee abandoned adding object-oriented features to C.

The remainder of this article will briefly list the different features that are in C99. Future articles in this series will describe individual features in greater detail.

Keywords

C99 has the following new keywords: **inline**, **restrict**, **_Bool**, **_Complex**, and **_Imaginary**.

The last three of the new keywords start with an underscore followed by an upper case letter in order to avoid conflicts with user identifiers in existing programs (**bool** in particular is common). However, the header **<stdbool.h>** defines a macro named **bool** that expands to **_Bool**, and the header **<complex.h>** defines a macro **complex** that expands to **_Complex**, and (if supported) a macro **imaginary** that expands to **_Imaginary**. The preferred style (once you determine it will not cause conflicts with identifiers in your program) is to include the appropriate header and use **bool**, **complex**, or **imaginary** rather than the underscore keywords. In the rest of this column, I will assume the proper headers have been included.

New Types

C99 has the following additional new types:

- **long long** and **unsigned long long**: integers with at least 64 bits.
- **bool**: a Boolean data type (with the same meaning as C++) that stores only 0 and 1. However, in order to make pointer arithmetic, arrays, and **sizeof** work, most implementations will store a non-bitfield **bool** in a byte rather than a bit.
- **float complex**, **double complex**, and **long double complex**: complex numbers corresponding to the three traditional floating-point types.
- **float imaginary**, **double imaginary**, **long double imaginary**: imaginary numbers corresponding to the three traditional floating-point types. The imaginary types are an optional part of the C Standard, and might not be commonly available.

Extended Integers

C99 allows implementations to define additional integer datatypes. All of the semantic rules dealing with integers in the C Standard were generalized to allow such "extended integers" to follow predictable rules and behave like any other integer type.

The new header **<stdint.h>** contains typedefs for integers of various sizes in bits (e.g., **int32_t**) or properties (like fast computation). The header also contains typedefs for the largest signed and unsigned integer types supported by the implementation, and (if such a type exists) a typedef for the integer type capable of holding the value of a pointer.

The header **<inttypes.h>** defines macros that are the **printf** and **scanf** format specifiers suitable for reading or writing values of all the different types named by typedefs in **<stdint.h>**.

Floating Point

The new header **<fpenv.h>** defines functions to allow you to control the floating-point environment, including rounding modes, status flags, and exception state.

The header **<math.h>** contains many new library functions. The new header **<complex.h>** contains math functions for complex numbers.

The new header **<tgmath.h>** defines type-generic function-like macros for many math functions (like intrinsic functions in Fortran or overloaded functions in C++). For example, after including **<tgmath.h>**, the call **sin(x)** will expand into a call to whichever sine function in the library takes an argument whose type is the same as the type of **x**.

C99 provides an optional specification of exactly how C behaves on a machine with IEEE floating-point arithmetic, including the rules for handling infinities, NaNs, signed zeroes, conversions, and expression evaluation.

C99 supports a new hexadecimal floating-point constant, which allows floating-point constants to be written without any loss of accuracy due to the decimal-to-binary conversion of traditional floating-point constants.

The header **<float.h>** contains additional information about the implementation.

New standard pragmas allow control of certain aspects of expression evaluation.

Arrays

The bounds of an array can now be a run-time expression; such arrays are called "variable length arrays," or VLAs for short. VLAs may not have static storage duration, and thus may not be declared at file scope, but they may be function parameters or local to a function. If local to a function, the correct amount of space for a VLA is allocated when the block containing the array is entered and the declaration of the VLA is reached. The storage is deallocated when leaving the block.

The last member of a struct may be an array with no bounds expression, called a flexible array member. If such a struct is allocated using **malloc**, the programmer can request additional storage to allow the flexible array member to be an array of any desired size.

Type qualifiers may appear after the **[** in the declaration of an array parameter to a function. Once the compiler has changed the type of the parameter from array to pointer, the type qualifiers modify the pointer type.

Features From C++

// comments are in C99.

Declarations do not have to appear at the start of a block. They may be intermixed with executable statements.

The **if**, **switch**, **while**, **do**, and **for** statements are now all blocks, as if a **{** preceded the statement and a **}** followed it. The first expression in a **for** statement (the initialization expression) may now be a declaration of the loop variable, which has scope of just the **for** statement.

Type qualifiers (**const**, **volatile**, **restrict**) may be redundantly specified.

Convenience Features

The enumerator list in the declaration of an enum type may have a trailing comma.

The minimum translation limits have been increased. Compilers are required to translate more complex programs.

Implementations must now support mixed-case external names. (C89 permitted implementations to force all external names to all upper case or all lower case.)

Optimization Features

Functions can be declared inline to encourage the implementation to eliminate function call overhead by inline substituting the body of the function at a call site. Inline functions may be extern.

A pointer can be declared with the **restrict** keyword. For example:

```
int *restrict p;
```

tells the optimizer that the pointer **p** is the only way to access the object to which **p** points. This potentially permits the compiler to produce much better code.

The keyword **static** may appear after the **[** in the declaration of an array parameter. This tells the optimizer that the array really is as big as specified, and may permit better code to be generated.

Initialization

A new feature, designated initializers, permits you to name the particular member or array element being initialized. For example:

```
struct S {
  int i;
  float f;
  int a[2];
};

struct S x = {
  .f=3.1,
  .i=2,
  .a[1]=9
};
```

Compound literals permit you to create an anonymous object and initialize it anywhere the value of such an object could appear. Syntactically, a compound literal is a cast followed by a brace-enclosed initializer. For example, if **f** is a function that takes an argument of type struct **S** above, you could write:

```
f((struct S) {2, 3.1, {0,9}});
```

Preprocessor

Macros may take a variable number of arguments. In the macro body, the special identifier **__VA_ARGS__** expands into a list of the variable arguments.

A macro argument may consist of no tokens.

C99 has a preprocessor operator of the form:

```
_Pragma ( string-literal )
```

This pragma operator behaves exactly as if a normal **#pragma** directive was encountered with the value of the string literal as its argument. However, the **_Pragma** operator may appear anywhere (not just at the beginning of a line) and macro bodies may contain **_Pragma**.

There are additional predefined macro names indicating the version of the C Standard supported, which optional parts of the C Standard are supported, and whether the implementation is hosted or freestanding (whether there is an operating system and C library the program can call).

Every function has the following implicit local variable:

```
static const char __func__[]
= "function-name";
```

where **function-name** is the name of the function. (Actually, **__func__** does not exist unless referenced in the function.) The **assert** macro uses **__func__** to report the function containing a failing assertion. (This is not a preprocessor feature, but it is similar to **__FILE__** and **__LINE__**.)

Preprocessor arithmetic is performed in the largest signed and unsigned integer types the implementation supports.

Program Correctness

C89 assumes an implicit type of **int** when a type is needed but never specified. This might happen when a variable is declared without a type, or a function does not have a declared return value. C99 requires a diagnostic be issued for these cases. Most implementations will issue a warning message, and then assume **int** in order to avoid breaking programs that relied on implicit **int**.

C99 requires a diagnostic if a return statement fails to return a value in a non-void function. It also requires a diagnostic if a return statement returns a value in a void function.

Internationalization

Accented letters, Non-English letters, and ideograms from languages like Chinese may be used in identifier names, including external identifiers.

The ISO 10646 Standard [5] is a universal character set whose goal is to have character codes for all characters in all languages. ISO 10646 has both two-byte and

four-byte character codes, and is a superset of Unicode. C99 permits you to represent any character in ISO 10646 by **\u** followed by four hex digits or **\U** followed by eight hex digits, where the hex digits are the character code in ISO 10646 for the character. These constructs are called the Universal Character Names or UCNs. You may use UCNs in strings, character constants, or identifiers.

Additional functions to process multibyte characters and wide characters are in the library.

The header **<iso646.h>** contains macros for some operators in C that require trigraphs to be used in some character sets.

Digraphs that are synonyms for some trigraphs are provided.

Library

In addition to new functions mentioned elsewhere in this article, the library contains some new specialized forms of **printf** and **scanf**. All **printf** and **scanf** family functions support new format conversion specifiers.

The **strftime** function supports additional conversion specifiers.

A new function, **va_copy**, is added to **<stdarg.h>**. It makes a copy of the variable argument pointer.

Next Month

Next Month we'll take a look at C99's new **restrict** keyword, and examine what restricted pointers can do to improve performance.

References

[1] ISO/IEC 9899:1999, Programming Languages — C. 1999.

[2] ISO/IEC 9899:1990, Programming Languages — C. 1990.

[3] ISO/IEC 9899 Amendment 1, Programming Languages — C Integrity. 1995.

[4] X3/TR-17:1997, Numerical C Extensions. 1997.

[5] ISO/IEC 10646, Information technology — Universal Multiple-Octet Coded Character Set (UCS).

Randy Meyers *is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*